Lab3

实验报告

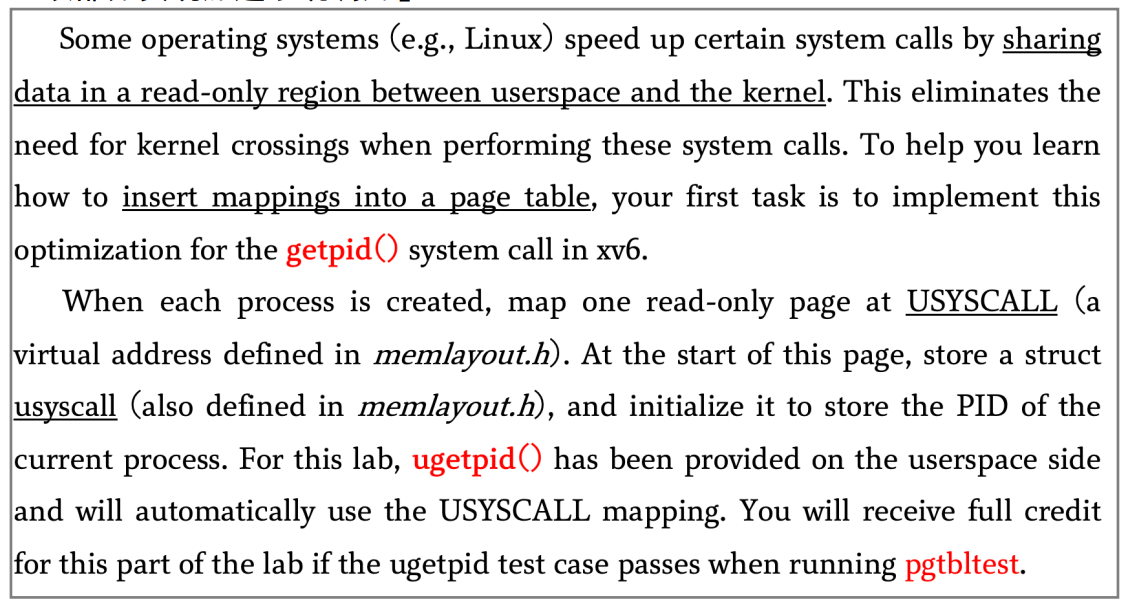
姓名 李晓畅

学号 20307130261

班级 计算机科学技术

# 实现思路

## Part A



* 考虑到需要和内核共享内存，在进程中添加一个包含内存地址的结构或许是个不错的选择。另外trapframe也是这样实现的。



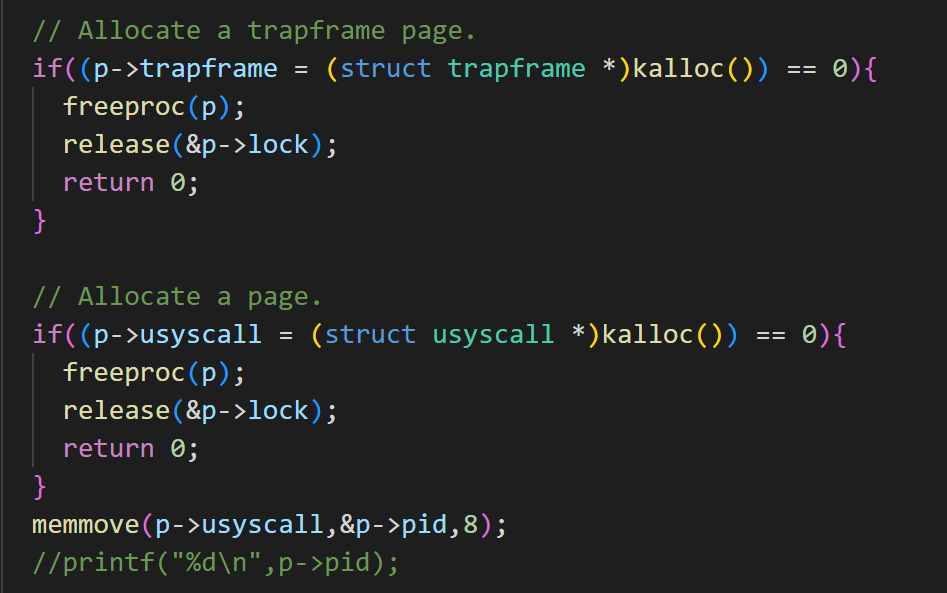
把它添加到***proc.h***中。

应该直接保存物理地址就好，否者还要多次映射反而可能不太好了。

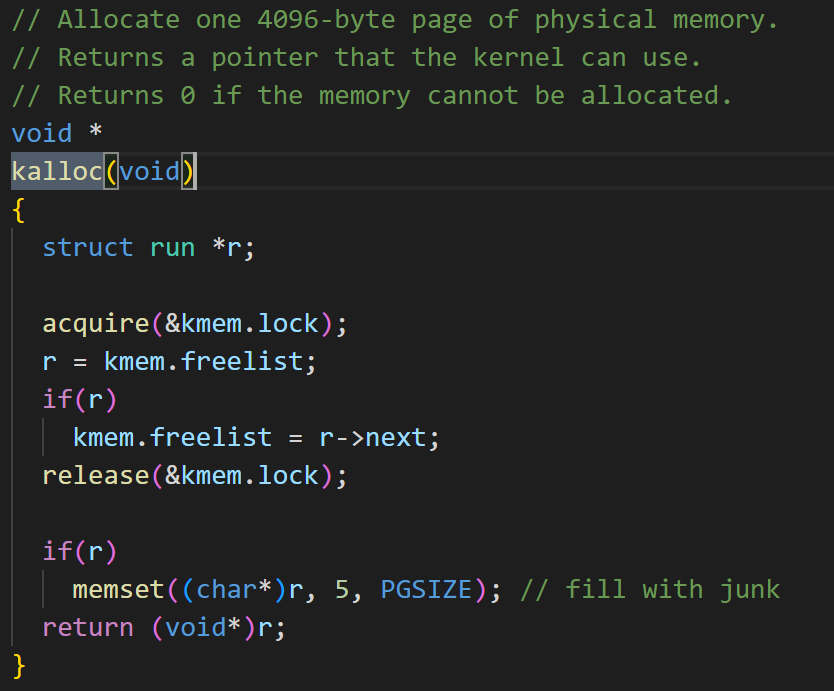
实际上我们之后会发现对应的函数也是这样要求的。

* 

因而在为进程创建分配资源的时候也应该为它对应分配资源



参照上面分配栈帧的写法就可以了。就是说分配一个块空间存储***pid***，如果失败就还原。这里不需要添加释放刚刚分配资源的部分，因为这会在***freeproc***中实现。我们只需要调用***freeproc***并释放锁就可以了。

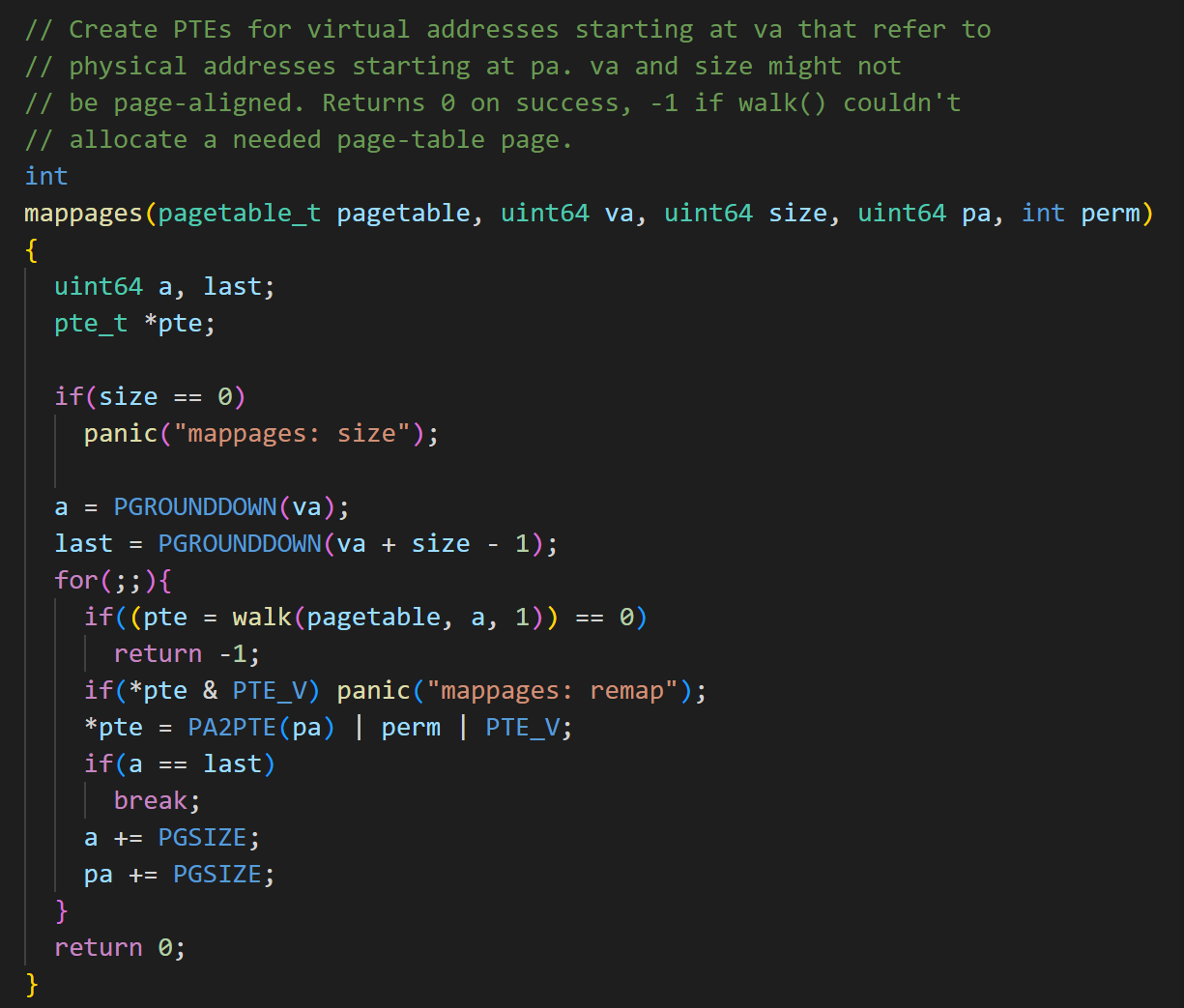


***Kalloc***用于分配一个页，返回的是物理地址。另外我们需要类型转换为对应的指针才可以赋值。

最后将***pid***拷贝到对应位置，然后就可以访问到了。

* 

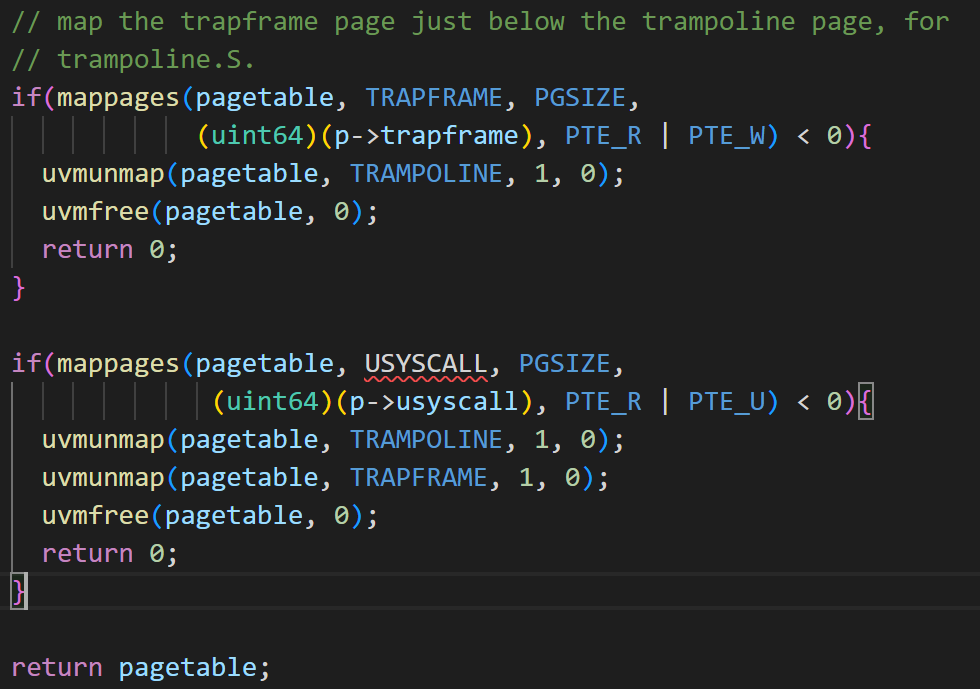
所以在创建用户页的时候也应该增加在内核中共享内存页的初始化，以及对共享内存块的页表初始化



有必要先看看***mappages***是干什么的，具体来说：

* 首先将要映射的虚拟地址页对齐。
* 将要映射的大小也页对齐，然后计算映射的结尾地址。
* 通过***walkpgdir***获得一个***pte***入口。
* 检查次***pte***是否已经被使用。
* 设置***pte***条目。
* 检查是否映射结束。
* 更新虚拟地址和物理地址，然后继续上述步骤。

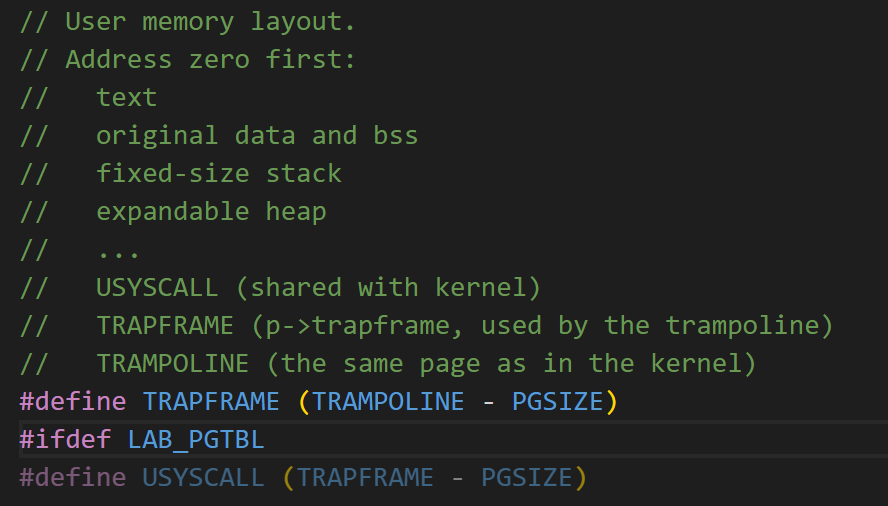
总之就是映射物理页到逻辑页。



实现方法还是和栈帧类似。因为不希望修改上面的分配代码，所以把它写在最下面，并且在分配失败时应该回收所有分配的空间。



选在USYSCALL参照***memlayout***中的说明。

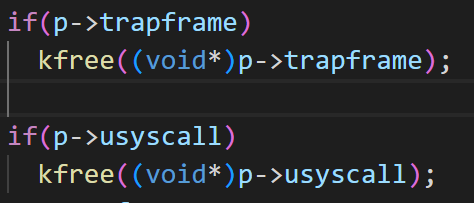


因为要和内存共享，应该要映射在***USYSCALL***中

* 

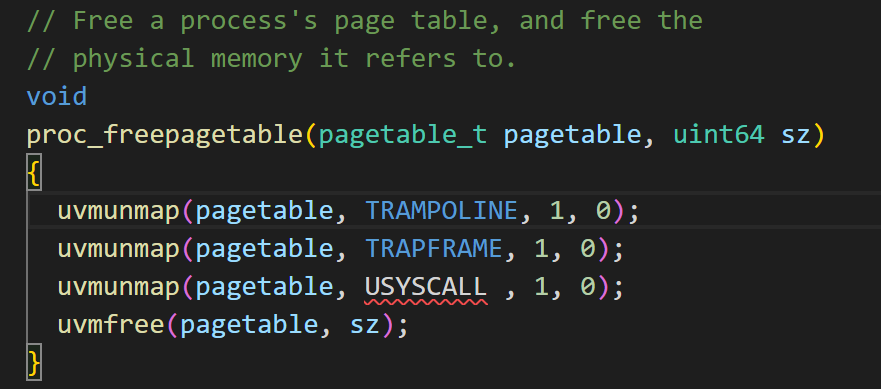
同上面分配空间相对应的是释放空间。

一方面是要释放为进程额外分配的空间：

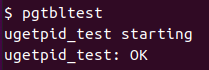


在***freeproc***中参考释放栈帧实现，释放。

另一方面是释放分配页时额外分配的空间：

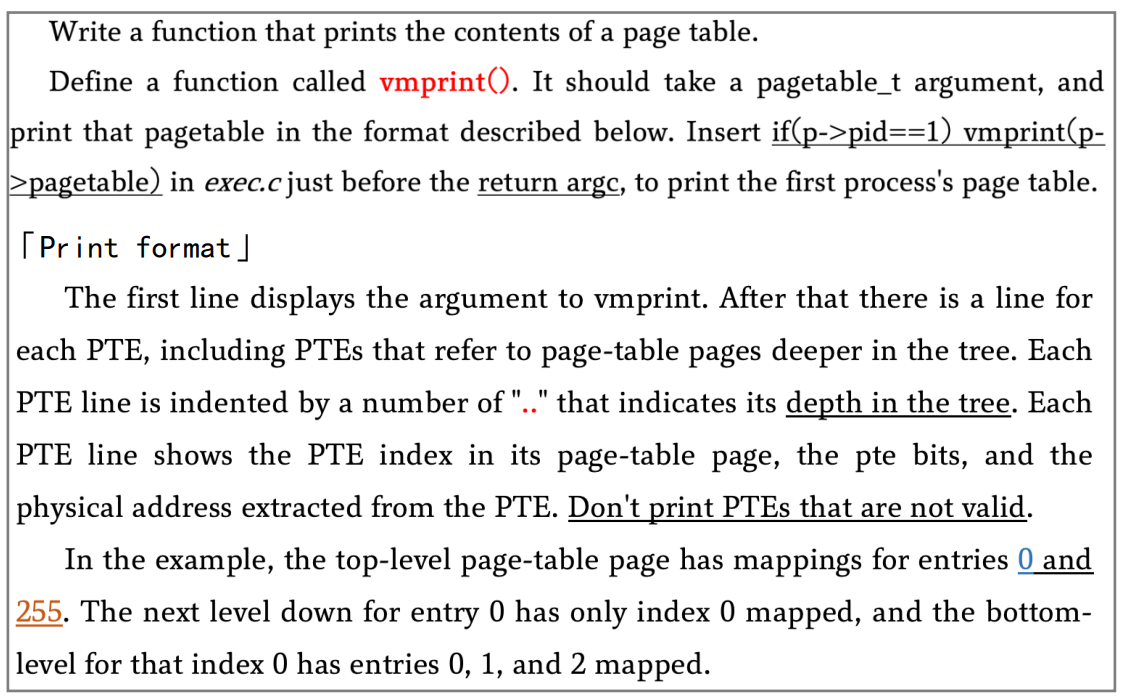


测试结果截图：

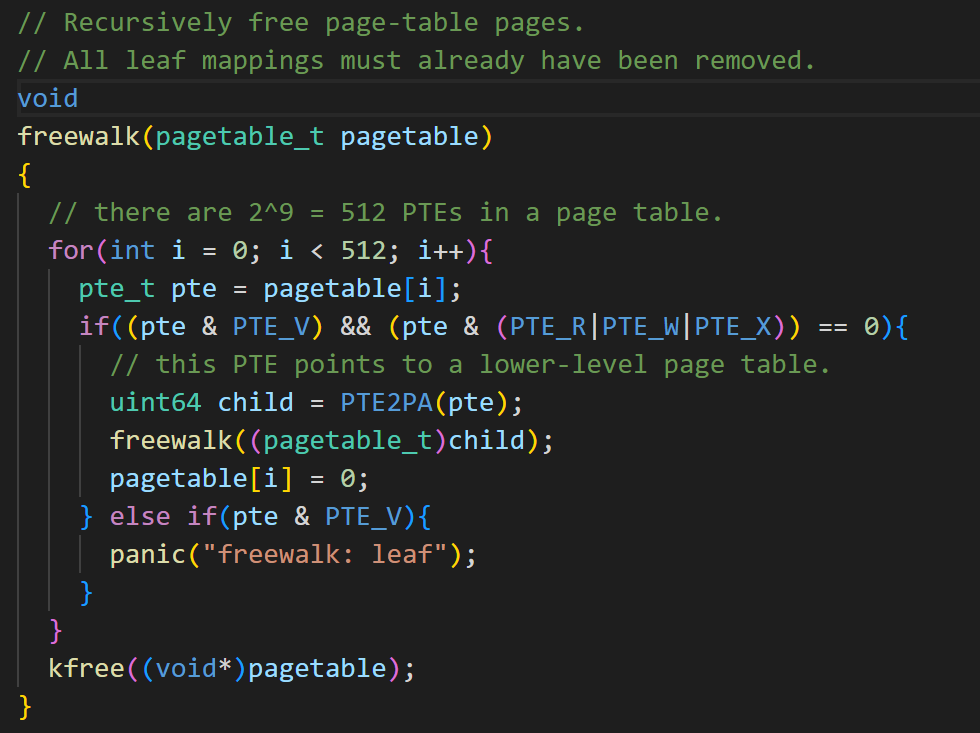


运行成功

## Part B



* 不妨先看看建议参考的函数

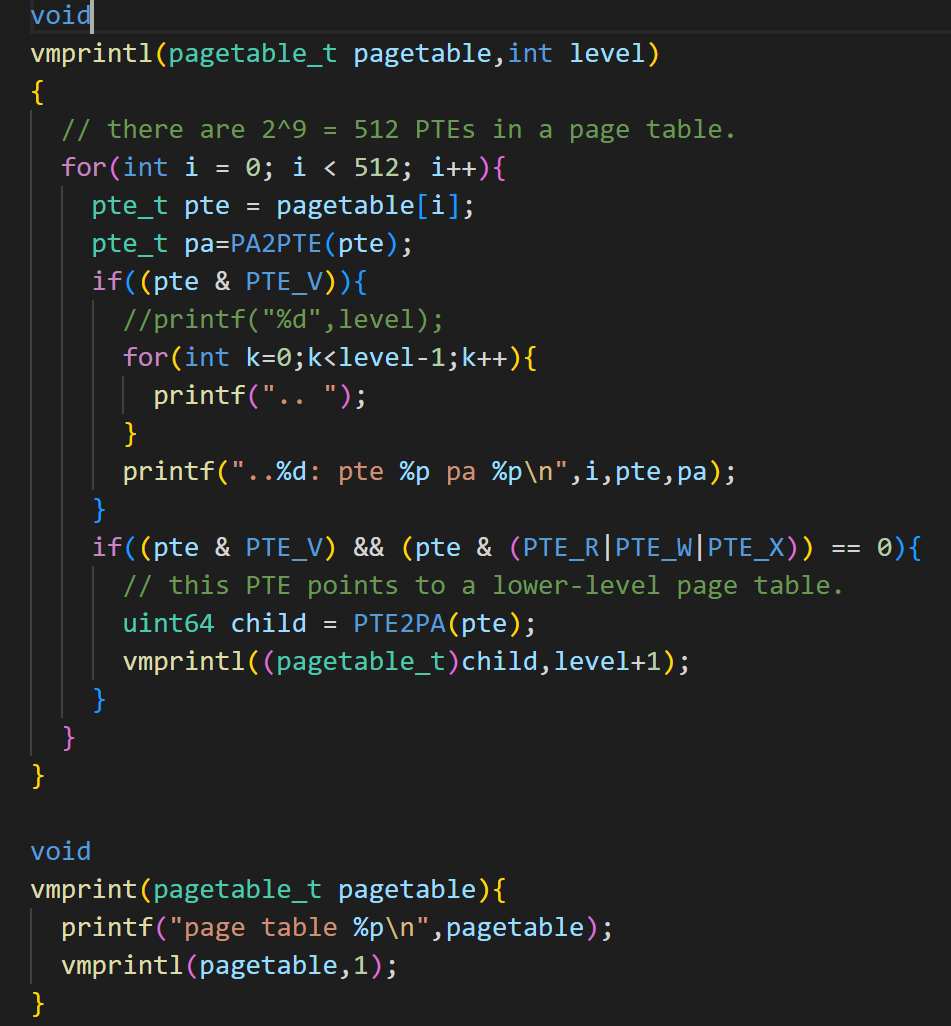


总之就是遍历页表中所有可用的PTE，对于非叶子的节点就要进入下一级页表映射，直到遍历每一层的所有PTE。

 ***freewalk*** 中判断到达叶节点的方式是检查 ***PTE\_R***, ***PTE\_W*** 或 ***PTE\_X*** 是否至少有一个被置位了，如果都没有那说明不是叶节点。这样判断的依据是，中间节点标志位的设置是在 ***walk*** 函数中进行的， ***walk*** 函数只给 PTE\_V 置位了，其余标志都是 0；而叶节点，一定会包含 ***PTE\_R***, ***PTE\_W***, ***PTE\_X*** 中的至少一个，因为实际的页应该至少要能读或写或执行，不然为什么要创建。

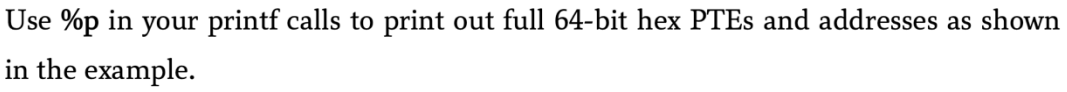
其实看完这个函数我们要如何实现就很显然了。

* 



因为每层打印的结果都不同，我们需要传递层数作为参数；但是在调用***Vmprint***的时候我们希望打印所有PTE，所以可以用两个分别实现。

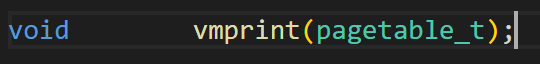
遍历表元，打印可用的PTE，如果不是叶子节点，就要递归调用函数打印其子节点。

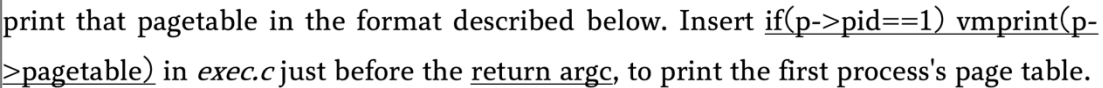


另外需要注意的是，打印PTE和地址应该用%p而非%d。

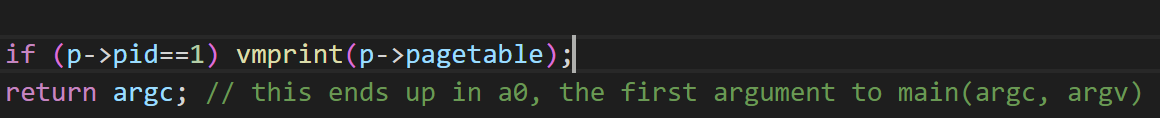
* 

为函数添加声明

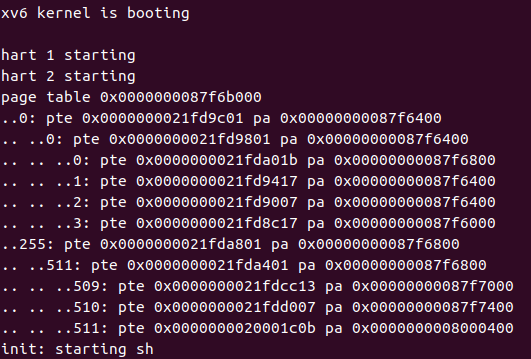


* 

修改第一个进程函数：

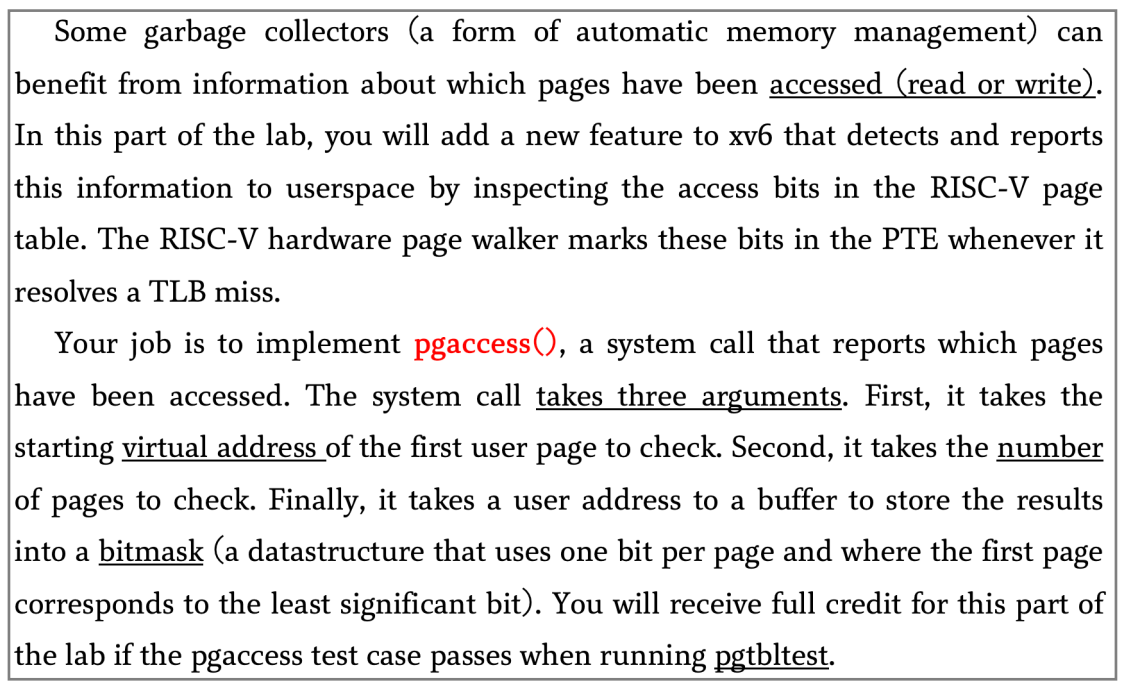


测试结果截图：

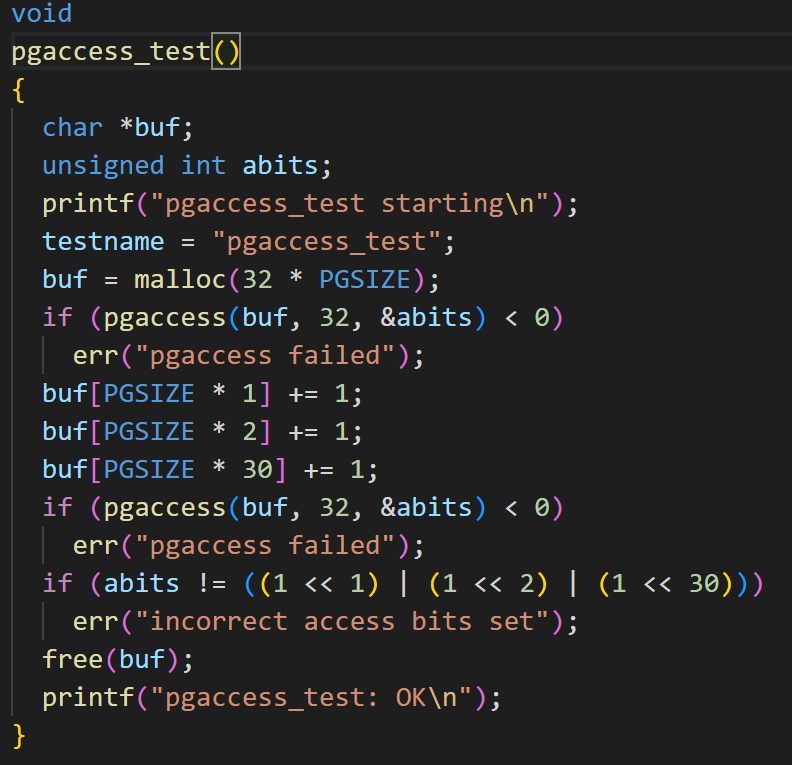


运行成功

## Part C



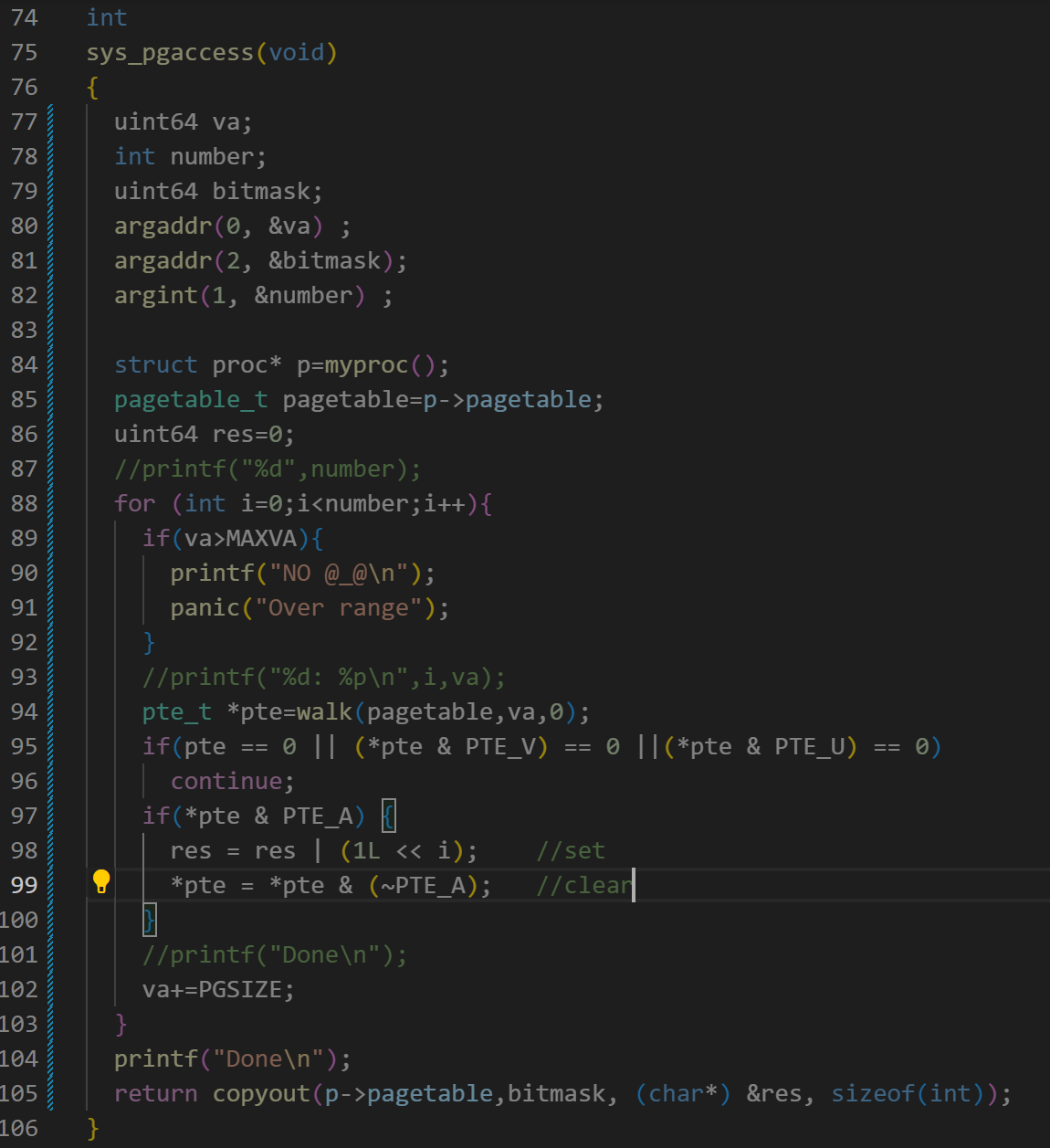
* 不妨先看看测试函数



大概就是分配空间，修改一些位置，检查被访问过的页是否被正确检测。

* 

其他配置其实已经完成了，所以直接实现就可以了。

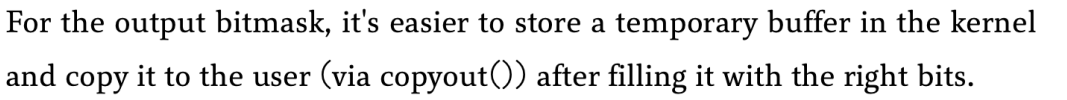


获取参数，然后遍历所有需要检查的逻辑页，映射到物理地址，跳过不可访问的页，对被访问过的页置位，拷贝结果。

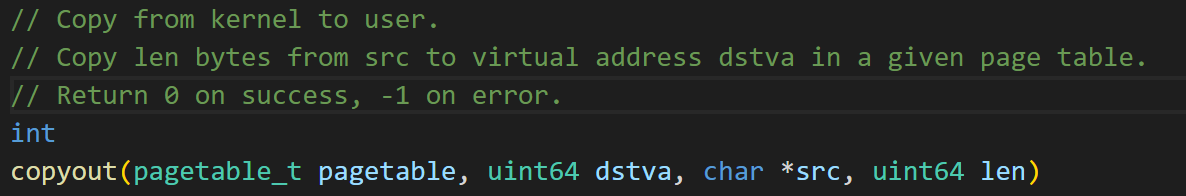


正如前一次实验中提到的那样，获取参数要用***argaddr***和***argint***。

***argint*** 利用用户空间的 ***%esp*** 寄存器定位第 n 个参数：***%esp*** 指向系统调用结束后的返回地址。参数就恰好在 ***%esp*** 之上***（%esp+4）***。因此第 n 个参数就在 ***%esp+4+4\*n***；***argaddr***也是类似的。

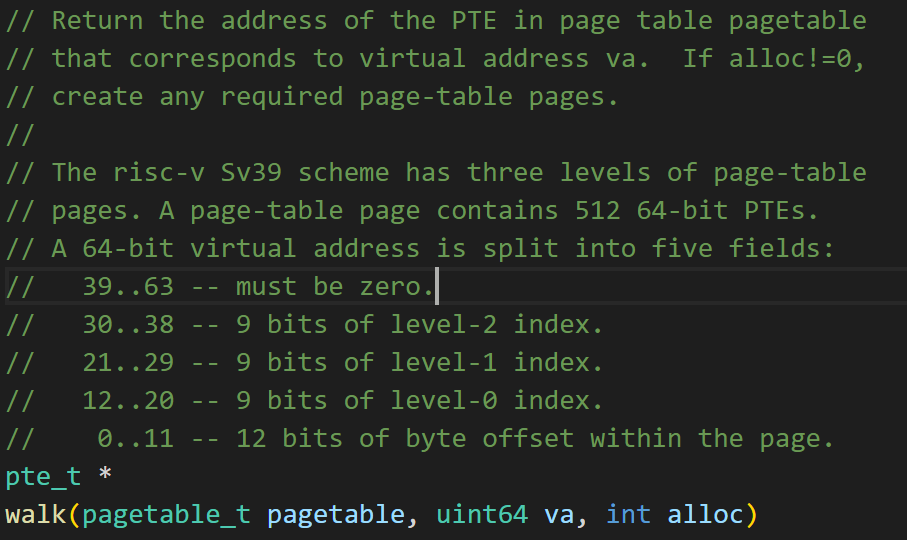


说得很对，我们应该维持一个缓冲，在结束检查后再写回用户态下的存储。

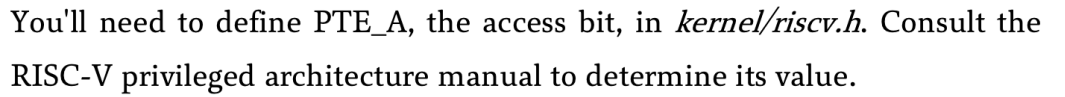


***Copyout***是把一块内核中的空间拷贝到一个逻辑内存指示的地方。注意参数类型。

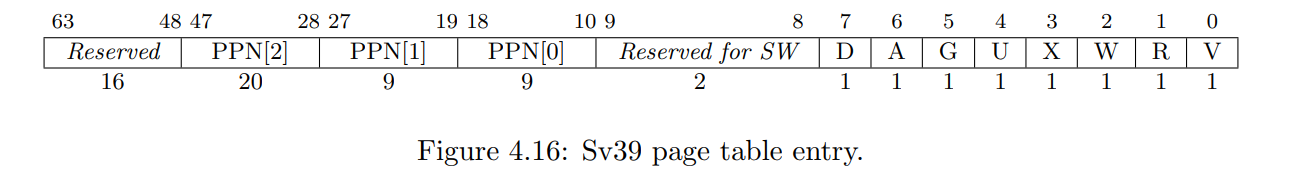


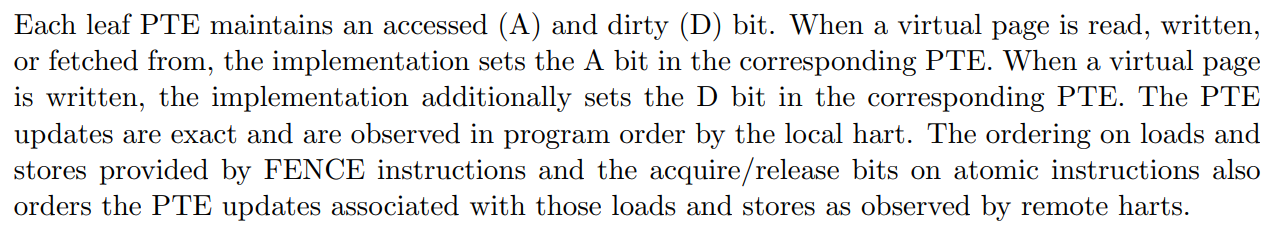


简单来说就是传入一级页表，把虚拟地址转换为实际PTE地址，然后这里不需要分配空间。



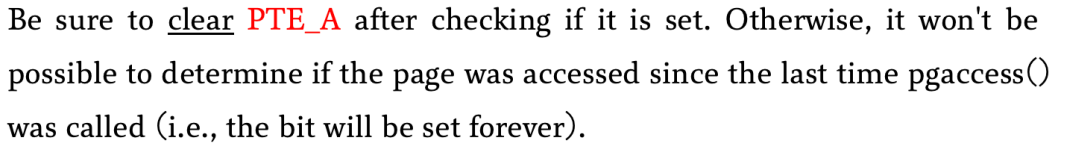
参考RISC-V privileged architecture:





所以access位应该是6，定义PTE\_A:



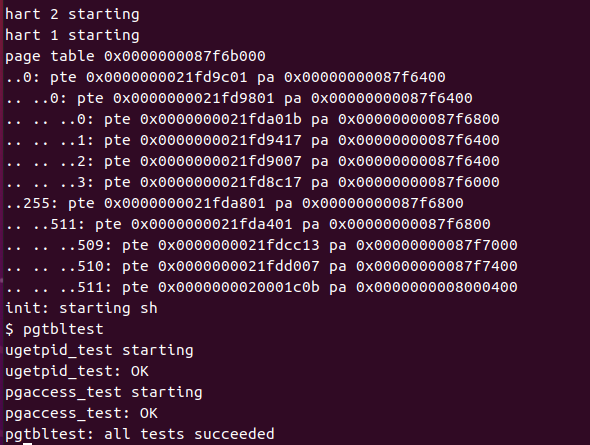


清除就置位为0就可以了。PTE\_A在前面已经定义过了。



应该是有用的吧。

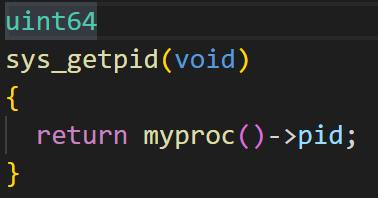
测试结果截图：



测试成功

# 问题回答

## 其他系统调用

观察sys\_getpid,不难发现这是一个系统调用，但是得到的是一个用户进程的参数，而且不会改变。这样共享内存就可以先在内核缓存对应的数值，使得内核可以直接读取参数，加速调用。

所以类似的还有：

* getppid, 获取父进程pid

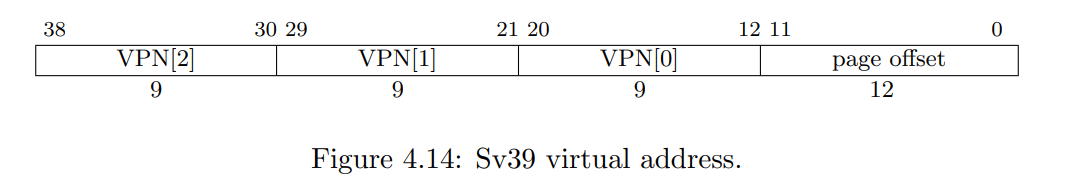
## 虚拟内存有什么用处？

* 高效使用内存：VM将主存看成是存储在磁盘上的地址空间的高速缓存，主存中保存热的数据，根据需要在磁盘和主存之间传送数据。这样不常用的内存就可以存储在磁盘中，使得内存被更好的利用；
* 简化内存管理：VM为每个进程提供了一致的地址空间，从而简化了链接、加载、内存共享等过程。这样就允许在程序运行时动态将逻辑地址映射到物理地址，内存管理更容易；
* 内存保护：保护每个进程的地址空间不被其他进程破坏。

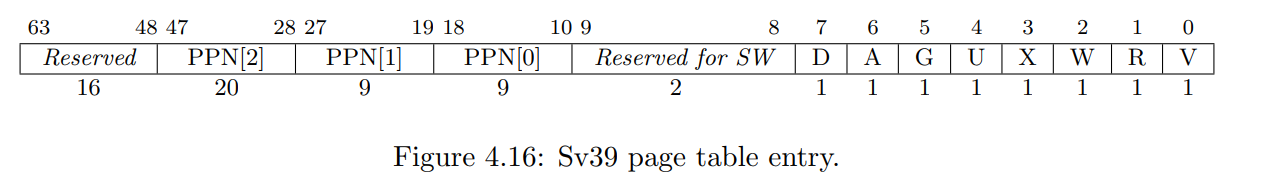
## 为什么现代操作系统采用多级页表？

* 使用多级页表可以使得页表在内存中离散存储。多级页表实际上是增加了索引，有了索引就可以定位到具体的项。为了快速随机访问，应该为一个页表分配一块连续的内存单元。这在范围不大时或许可以，但在范围较大时或许就不太理想。

比如考虑我们实验中用到的RISK-V系统：



39位地址空间中有27位表示页序号。换句话说，页表总计有2^27项。



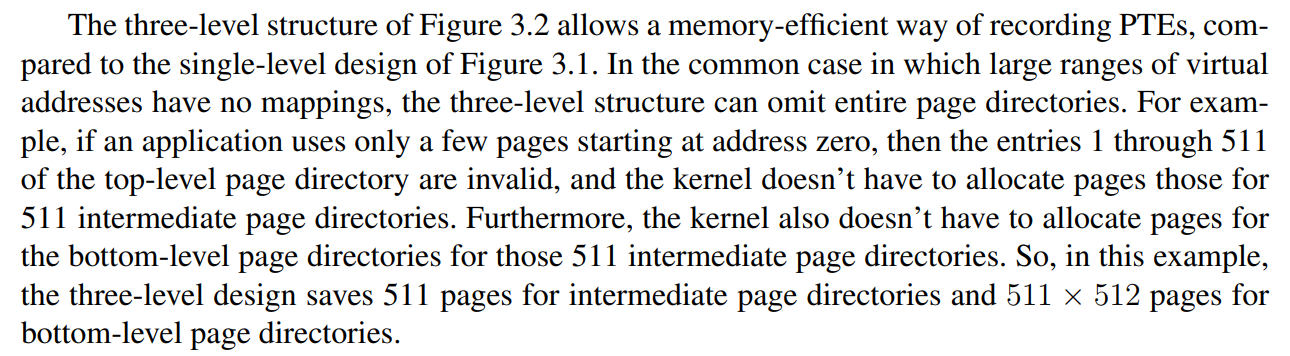
一个页表项需要64bit，换言之8B

总计需要：8B\*2^27=2^30B=1G

哪来那么多连续空间。

而多级页表就不需要连续了。连续也不会更快。

* 使用多级页表可以节省页表内存。使用一级页表，需要连续的内存空间来存放所有的页表项。多级页表通过只为进程实际使用的那些虚拟地址内存区请求页表来减少内存使用量。多级页表相当于一棵树，如果没有那么多的地址空间需要寻址，那么就没有必要建完全树了。就是这个例子：



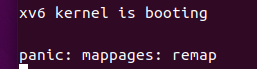
## Part C 的detect流程

* 申请空间，为它分配对应的页。
* 访问对应的页。硬件会负责为被访问过的页页表中的PTE\_A位置1。
* 找到要检测页的PTE，检查它的PTE\_A位的值，修改bitmask。
* 复位，这样下次还能检测。

# 问题和解决

## 调试和检查

在PART A中，会报这个错：

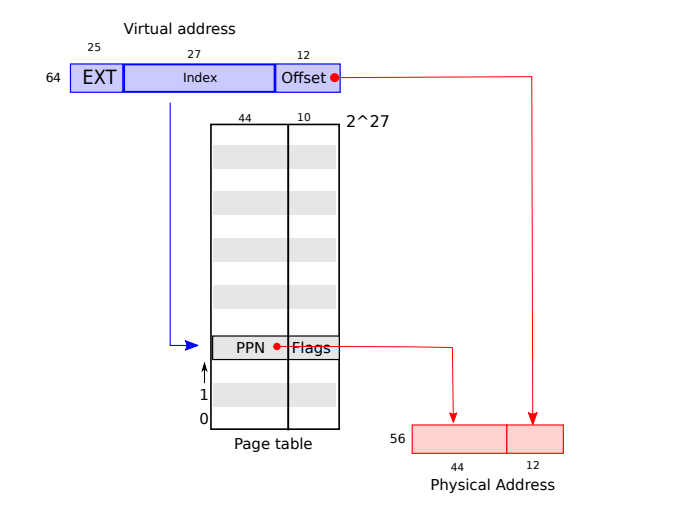


利用添加输出语句、GDB调试方法并加以分析，定位问题是重复映射。检查发现是一个参数写错了。对应修改即可。

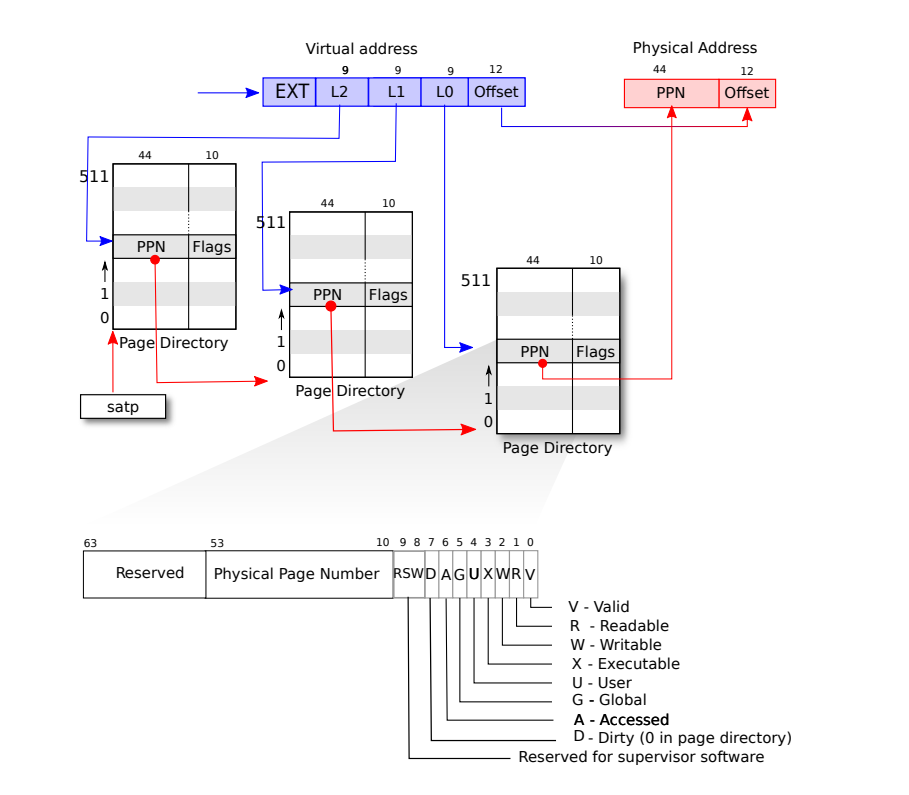
## 内存实现

在PART C中直接在页表中检查PTE\_A是不对的。

逻辑上说确实是这样：



但实际上是三层页表：



所以要参照建议使用walk函数，或者自己实现映射。

# 实验感想

## 数据结构的作用

在本次学习的页表中，深刻体现出灵活运用数据结构在提高效率方面的作用。简单的页表类似于map，比起直接管理物理内存，使用页表更加方便。多级页表类似于树，比起单级页表它有利于离散存储，节约空间。

## 类比

Xv6的函数不像很多库那样有非常完备的注释，因而理解它的作用或许需要观察它在其他函数中的使用来实现。参照其他地方编写映射、调用函数，这样同样可以实现目的。换句话说，我们理解代码的方式其实是多种多样的。